Solving the Maze by Depth-First Traversal

Linh Bien

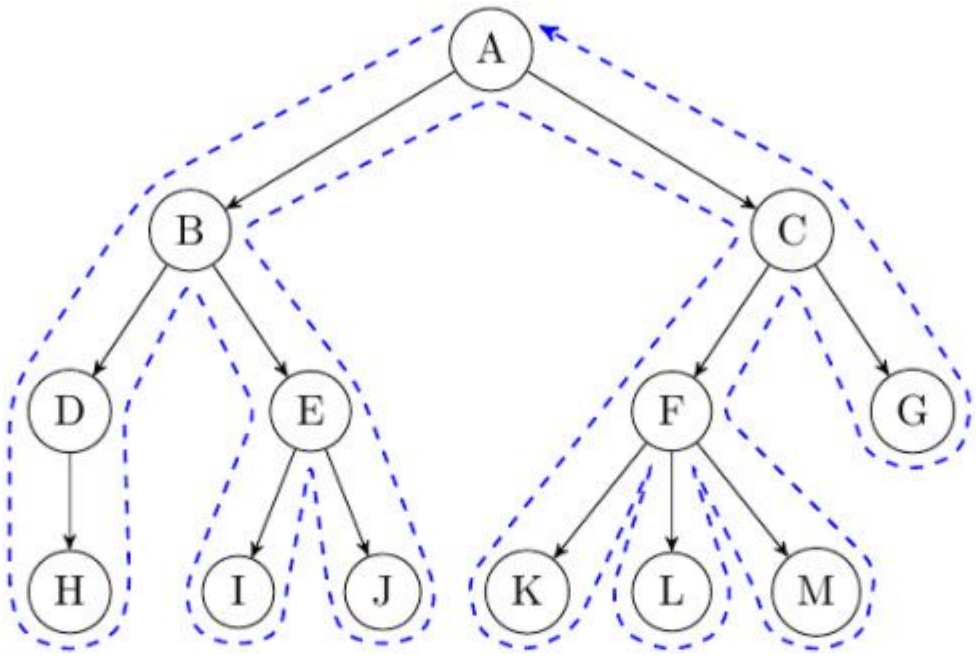
Table of content

Introduction

- Problem: Find if there is way to go from the start position to the destination position in the Maze
- Solution: Apply Depth First Traversal. Depth First Search (DFS) is algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration

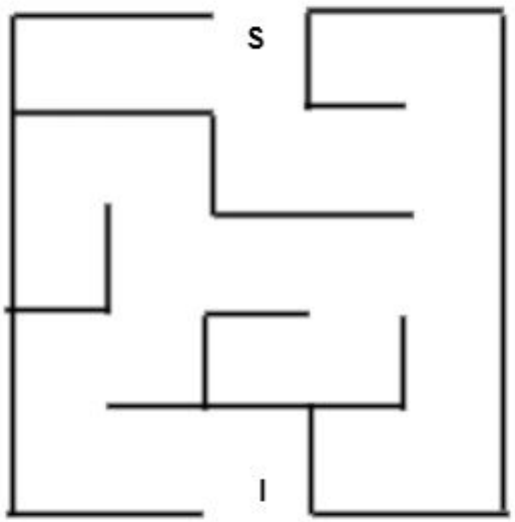
Implement

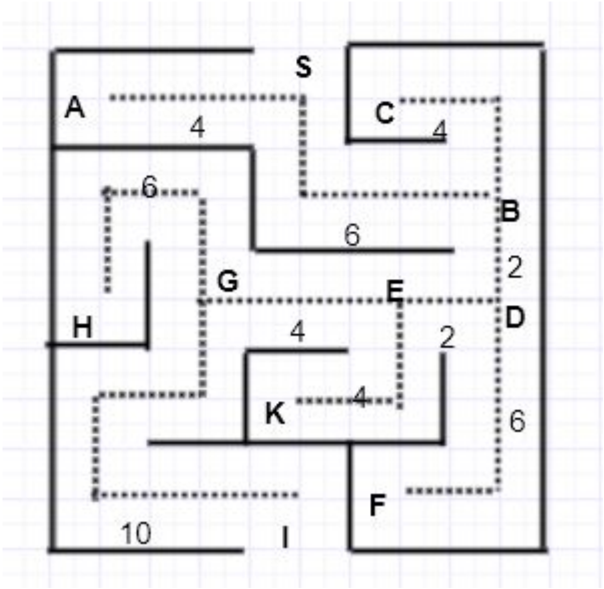
- **Rule 1**
  1. Visit the adjacent unvisited vertex.
     - If there are several vertices, randomly pick one.
  2. Mark it as visited.
  3. Display it.
  4. Push it in a stack.
- **Rule 2**
  1. If no adjacent vertex is found, pop up a vertex from the stack.
     - It will pop up all the vertices from the stack, which do not have adjacent vertices.
- **Rule 3**
  1. Repeat Rule 1 and Rule 2 until the stack is empty.



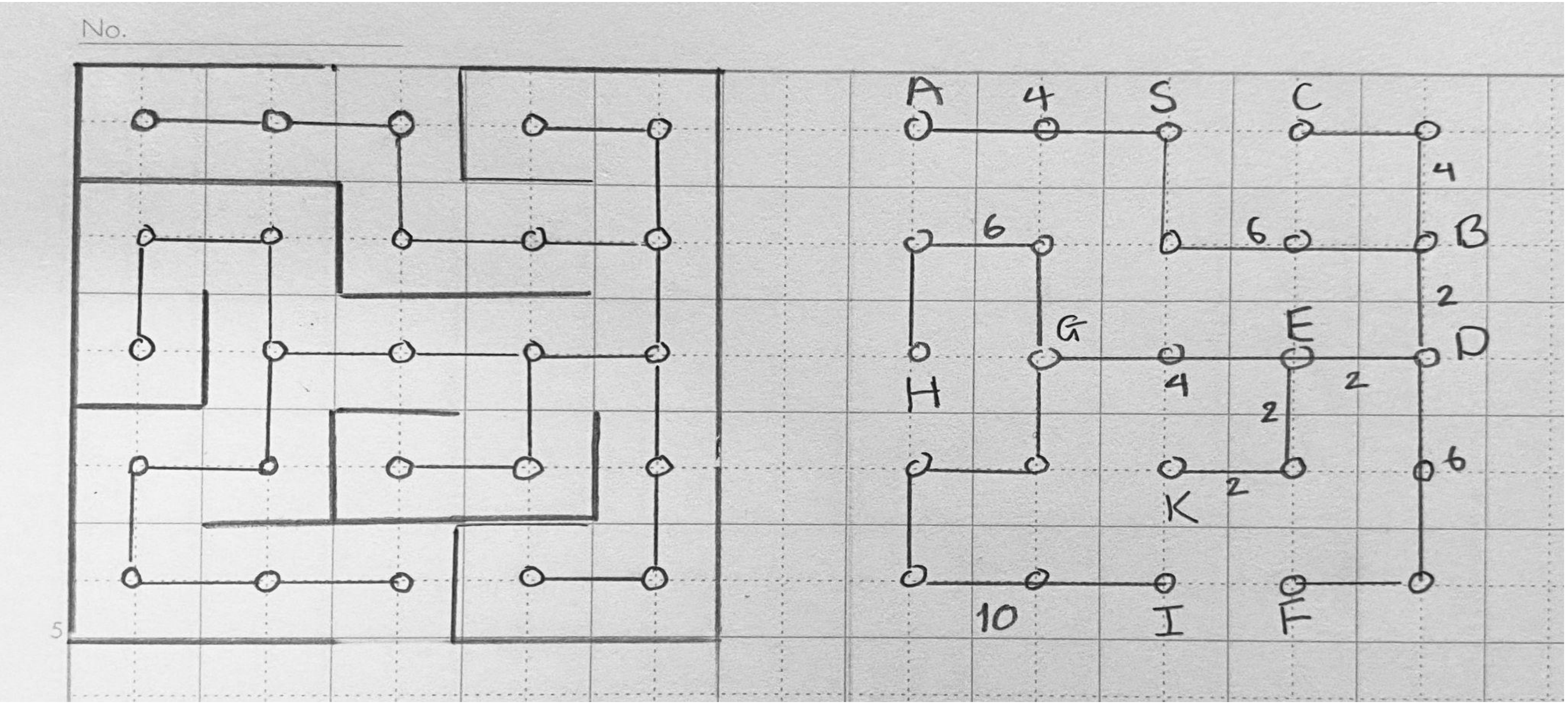
Example 1

Given a Maze: Return True if can go from start S to destination I. If not, return False





Example 1



Example 1

Let’s view in form of a tree. S is the node and 2 branches: right, left

From S we can go to A or B. A is dead end. Go to S, B

From B we can go to C or D. C is dead end. Go to B, D

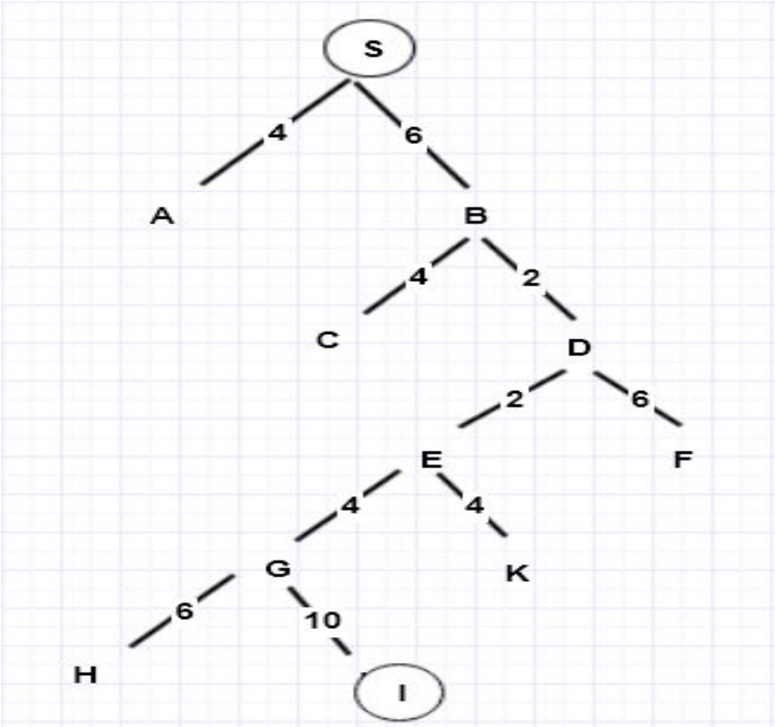
From D we can go to E or F. Go to E (F is dead end: Ignore)

From E we can go to G or K. Go to G. (K is dead end: Ignore)

From G we can go to H or I. H is not destination. Go to G, I. I is the destination.

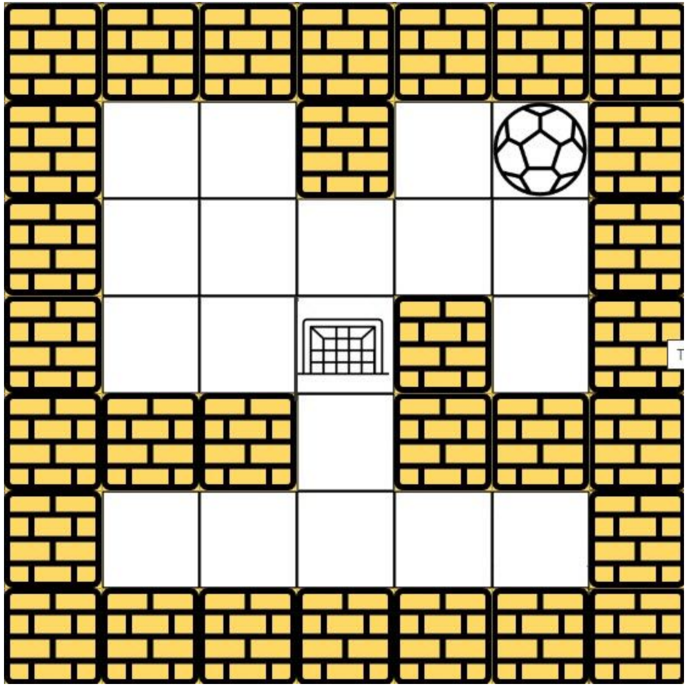
S -> B -> D -> E -> G -> I : True

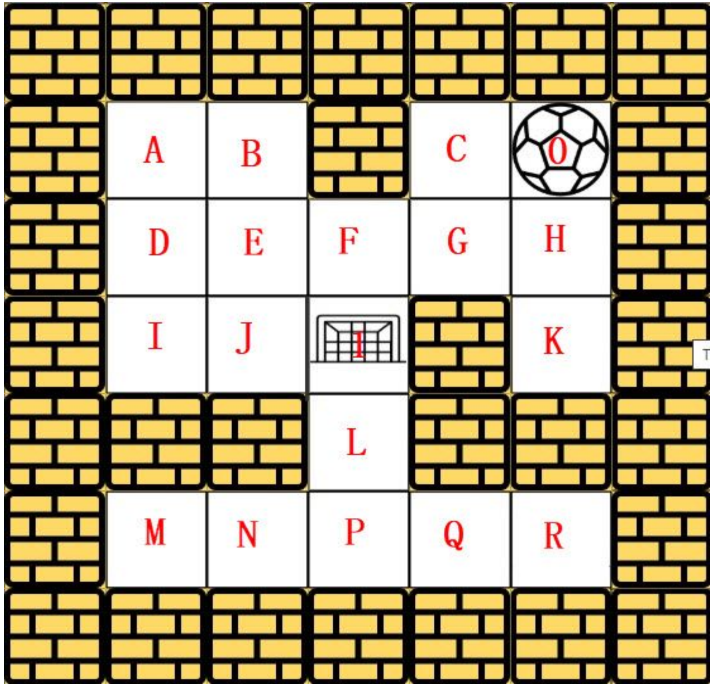
Distance: 24 units



Example 2

Conduct Depth_First Traversal - Right, Left, Up, Down

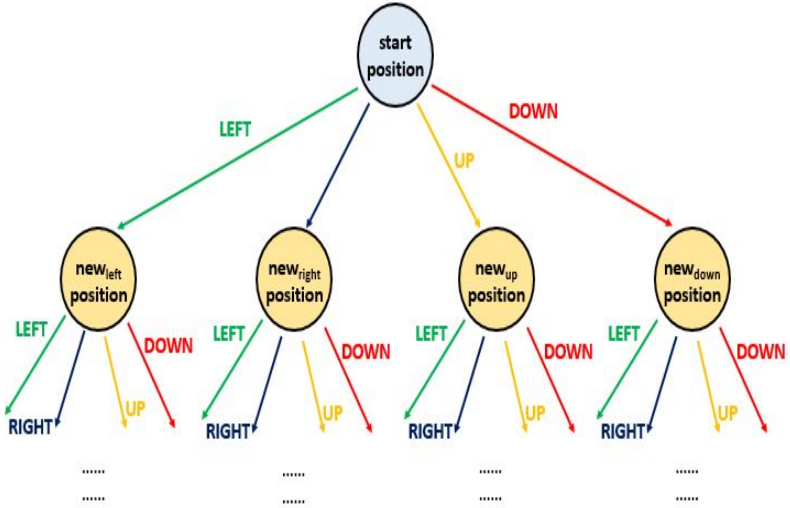




Example 2

Let’s view the given search space in a form of a tree:

- Starting position: the root node of the tree
- Right, left, up or down: 4 different routes, 4 branches
- The new node reached from the root traversing over the branch represents the new position occupied by the ball after choosing the corresponding direction of travel



Example 2

In a tree: 0 is the node, 4 branches: right, left, up, down

From 0 can go to C or K

From C can go to G

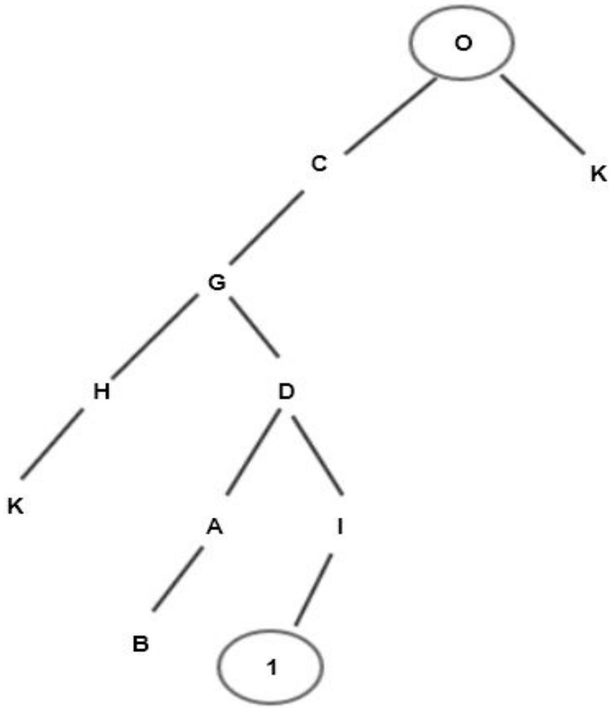
From G can go to D or H

From H can go to K (K is dead end). Go back to G -> D

From D can go to A or I (other points are visited)

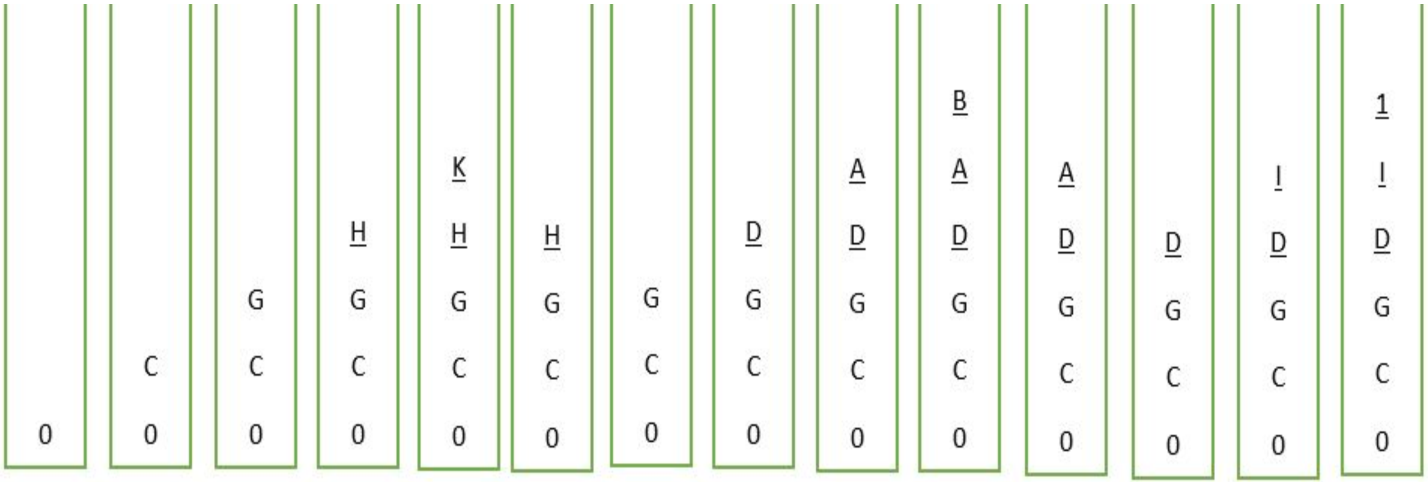
From A can go to B (B is dead en). Go back to A ->D -> I

From I can go to 1: destination



Example 2

Use stack:



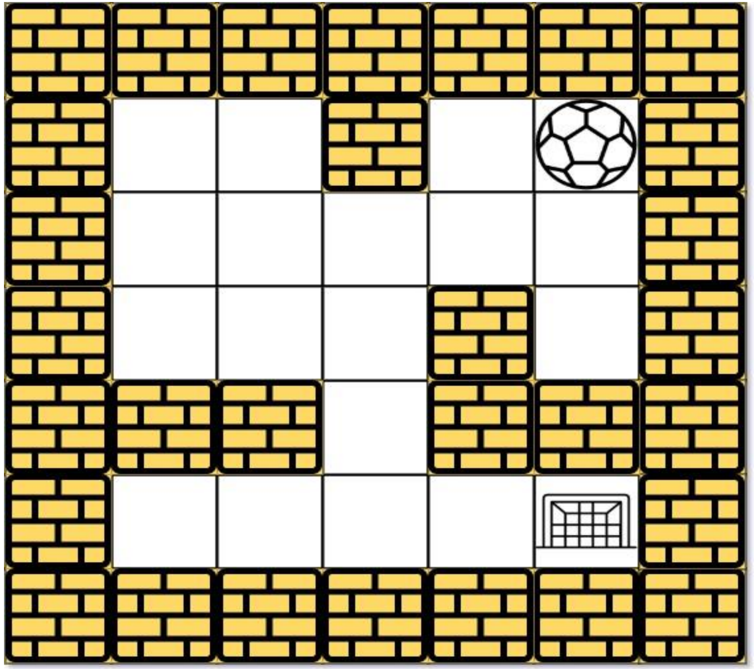
419. The Maze (Leetcode)

Description:

There is a ball in a `maze` with empty spaces (represented as `0`) and walls (represented as `1`). The ball can go through the empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the `m x n` `maze`, the ball's `start` position and the `destination`, where `start = [startrow, startcol]` and `destination = [destinationrow, destinationcol]`, return `true` if the ball can stop at the destination, otherwise return `false`.

You may assume that the borders of the maze are all walls



Programming

```
from typing import List
class Solution:
    def hasPath(self, maze: List[List[int]], start: List[int], destinatio
        directions = [(1,0),(-1,0),(0,-1),(0,1)]
        m = len(maze)
        n = len(maze[0])
        stack = []
        seen = set()
        stack.append((start[0], start[1]))
        seen.add((start[0], start[1]))
        while stack:
            curr_i, curr_j = stack.pop()
            for d in directions:
                i = curr_i
                j = curr_j
                while 0 <= i < m and 0 <= j < n and maze[i][j] ==0:
                    i += d[0]
                    j += d[1]
                i -= d[0]
                j -= d[1]
                if i == destination[0] and j == destination[1]:
                    return True
                if (i,j) not in seen:
                    stack.append((i,j))
                    seen.add((i,j))
        return False
```

```python
from typing import List
class Solution:
    def hasPath(self, maze: List[List[int]], start: List[int], destination: List[int]) -> bool:
        directions = [(1,0),(-1,0),(0,-1),(0,1)]
        m = len(maze)
        n = len(maze[0])
        stack = []
        seen = set()
        stack.append((start[0], start[1]))
        seen.add((start[0], start[1]))
        while stack:
            curr_i, curr_j = stack.pop()
            for d in directions:
                i = curr_i
                j = curr_j
                while 0 <= i < m and 0 <= j < n and maze[i][j] ==0:
                    i += d[0]
                    j += d[1]
                i -= d[0]
                j -= d[1]
                if i == destination[0] and j == destination[1]:
                    return True
                if (i,j) not in seen:
                    stack.append((i,j))
                    seen.add((i,j))
        return False
```

PROBLEMS 22 DEBUG CONSOLE OUTPUT TERMINAL JUPYTER Python

```
Output:  True
Input: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [3,2]
Output:  False
Input: maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]], start = [4,3], destination = [0,1]
Output:  False
PS C:\Users\bienh\OneDrive\Documents\CS summer tremester\Algorithms>
```

Ln 27, Col 1 Spaces: 4 UTF-8 CRLF Python 3.9.12 ('base': conda)

ENG 5:24 PM 7/17/2022

Test

```
def main():
    test = Solution()
    print("Input: [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [4,4]")
    print("Output: ",test.hasPath([[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]],[0,4],[4,4]))


    print("Input: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [3,2]")
    print("Output: ",test.hasPath([[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], [0,4],[3,2]))


    print("Input: maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]], start = [4,3], destination = [0,1]")
    print("Output: ",test.hasPath([[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]],[4,3],[0,1]))
main()
```

Output

Input: [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [4,4]

Output: True

Input: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [3,2]

Output: False

Input: maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]], start = [4,3], destination = [0,1]

Output: False

Conclusion

- **Depth First Search (DFS) is an algorithms traverses a graph in a depthward motion**
- **We can view the given space search in a form a a tree with node as the start position and branches are all the different options(For examples: 2 options: right, left or 4 options: right, left, up, down)**
- **Use stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration**

Enhancement ideas

- **Depth - first search (DFS) is useful in cycle detection in graphs, and solving puzzles with only one solution such as a maze or a sudoku puzzle**
- **DFS is also used in mapping routes, scheduling and finding spanning trees**
- **DFS consumes less memory space and will reach at the goal node in a less time if it traverses in a right path.**

References

Karleigh M, Ken J, Jimin K, Depth-First Search (DFS). Retrieved 07/18/2022 from https://brilliant.org/wiki/depth-first-search-dfs/#complexity-of-depth-first-search

Tutorialspoint. Data Structure - Depth First Search. https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm